19장 프로토타입

객체지향 프로그래밍

전통적인 명령형 프로그래밍의 절차지향적 관점에서 벗어나 객체의 집합으로 프로그램을 표현하는 프로그래밍

객체

실체 곧 객체는 **특징이나 성질**을 나타내는 **속성**을 가지며 인식하거나 구별할 수 있다

객체

- 속성을 통해서 여러개 의 값을 하나의 단위로 구성한 복합한 자료구조

- 상태 데이터와 동작을 하나의 논리적인 단위로 묶는 복합적 자료구조
  - 객체의 상태 = 데이터
  - 객체의 동작 = 데이터를 조작

추상화

다양한 속성 중 에서 필요한 속성만 간추려 표현하는것 을 **추상화**라 칭한다

상속과 프로토타입

상속 : 어떤 객체의 프로퍼티 또는 메서드를 다른객체가 상속받아 그래도 사용할 수 있는것

프로토타입을 사용하지 않을 시

```
function Circle(radius) {
  this.radius = radius;
  this.getArea = function () {
    return Math.PI * this.radius ** 2;
  };
}
```

```
const circle1 = new Circle(1);
const circle2 = new Circle(2);

console.log(circle1.getArea === circle2.getArea);   // false
console.log(circle1.getArea()); // 3.14159...
console.log(circle2.getArea()); // 12.56637...
```

getarea 메서드를 중복 생성하고 중복 소유한다 → 메모리 낭비

자바스크립트는 프로토타입을 기반으로 상속을 구현하여 불필요한 중복을 제거한다

```
// 생성자 함수
function Circle(radius) {
  this.radius = radius;
}

// Circle 생성자 함수가 생성한 모든 인스턴스가 이 메서드를 사용할 수 있도
// 프로토타입에서 관리한다.
// 프로토타입은 Circle 생성자 함수의 prototype 프로퍼티에 바인딩 되어있

Circle.prototype.getArea = function () {
  return Math.PI * this.radius ** 2;
};
// 인스턴스 생성
const circle1 = new Circle(1);
const circle2 = new Circle(2);
// 모든 인스턴스는 하나의 getArea 메서드를 공유한다
console.log(circle1.getArea === circle2.getArea); // true

console.log(circle1.getArea()); // 3.14~
console.log(circle2.getArea()); // 12.56~
```

Circle 생성자 함수가 생성한 모든 인스턴스는 자신의 프로토타입, 즉 상위 객체 역할을 하는 Circle.prototype의 모든 프로퍼티와 메서드를 상속받는다.

고로 getarea 메서드는 단 하나만 생성되어 상속되어 사용되고

radius 프로퍼티만 개별적으로 소유한다

프로토타입 객체

프로토타입 객체(또는 프로토타입)란 객체 간 상속(inheritance)을 구현하기 위해 사용된다

객체가 생성될 때 객체 생성 방식에 따라 프로토타입이 결정되고 [[Prototype]]에 저장된다.

저장되는 프로토타입은 객체 생성 방식에 따라 결정된다

1. 객체리터럴 → Object.prototype
2. 생성자함수 →생성자함수의 프로토 타입

모든 객체는 하나의 프로토타입을 가지며 , 모든 프로토타입은 생성자함수와 연결되있다.

프로토타입은

constrcutor를 통해 생성자함수 에 접근

생성자함수는

prototype을 통해 프로토타입 에 접근

__proto__ 접근자 프로퍼티

[[Prototype]] 내부 슬롯에는 직접 접근할 수 없다.

대신, **proto** 접근자 프로퍼티를 통해 자신의 프로토타입에 간접적으로 접근할 수 있다.

- 접근자 프로퍼티
  - 접근자 프로퍼티는 자체적인 값 프로퍼티를 가지지 않고

    접근자 함수(get,set)로 구성된 프로퍼티 이다

- 상속을 통해 사용
  - **proto** 접근자 프로퍼티는 객체가 직접 소유하는 프로퍼티가 아니라 Object.prototype의 프로퍼티다.

- 접근자 프로퍼티를 통해 접근하는 이유

```
const parent = {};
const child = {};

child.__proto__ = parent;
parent.__proto__ = child; // ❌ TypeError: Cyclic __proto_
```

  - 상호 참조에 의해 프로토타입 체인이 생성되는 것을 방지하기 위해서다.
  - 프로토타입 체인은 단방향 링크드 리스트로 구현되어야 한다.
  - 순환 참조하는 프로토타입 체인이 만들어지면 프로토타입 체인 종점이 존재하지 않기 때문에 프로토타입 체인에서 프로퍼티를 검색할 때 무한 루프에 빠진다.

- **__proto__** 사용
  - *proto_* 접근자 프로퍼티를 코드 내에서 직접 사용하는 것은 권장하지 않는다. 모든 객체가 **proto** 접근자 프로퍼티를 사용할 수 있는 것은 아니기 때문 (ex) object.create(null))
  - **proto** 대신 Object.getPrototypeOf 메서드를 사용하여 프로토타입의 참조를 취득하고 Object.setPrototypeOf 메서드를 사용해서 프로토타입을 교체하는 것을 권장

함수 객체의 prototype

함수 객체만이 소유하는 prototype 프로퍼티는 생성자 함수가 생성할 인스턴스의 프로토타입을 가리킨다.

생성자 함수로서 호출할 수 없는 함수,

즉 non-constructor인

1. 화살표 함수
2. ES6 메서드 축약 표현으로 정의한 메서드

는 prototype 프로퍼티를 소유하지 않으며 생성하지도 않는다.

구분	소유	값	사용 주체	사용 목적
__proto__ 접근자 프로퍼티	모든 객체	프로토타입의 참조	모든 객체	객체가 자신의 프로토타입에 접근 또는 교체하기 위해 사용
prototype 프로퍼티	constructor	프로토타입의 참조	생성자 함수	생성자 함수가 자신이 생성할 객체의 프로토타입을 할당하기 위해 사용

prototype의 constructor 프로퍼티

모든 프로토타입은 constructor 프로퍼티를 갖는데, 자신을 참조하고 있는 생성자 함수를 가리킨다.

리터럴 표기법에 의해 생성된 객체의 생성자 함수와 프로토타입

리터럴 표기법에 의해 생성된 객체의 경우 프로토타입의 constructor 프로퍼티가 가리키는 생성자 함수가 반드시 객체를 생성한 생성자 함수라고 단정할 수는 없다.

리터럴 표기법	생성자 함수	프로토타입
객체 리터럴	Object	Object.prototytpe
함수 리터럴	Function	Function.prototype
배열 리터럴	Array	Array.prototype
정규 표현식 리터럴	RegExp	RegExp.prototype

리터럴표기법의 객체 생성시 생성자 함수와 프로토타입

리터럴 표기법으로 객체를 생성시 에도 내부적으로 가상적인 생성자 함수를 갖고

OrdinaryObjectCreate를 호출하여 Object.prototype를 프로토타입으로 갖는 빈 객체를 갖는다

프로토타입은 생성자 함수와 더불어 생성되며 prototype,constructor 프로퍼티에 의해 연결되어잇다

결론적으로

프로토타입과 생성자 함수는 단독으로 존재할 수 없고 언제나 쌍으로 존재한다.

프로토타입의 생성 시점

프로토타입은 생성자 함수가 생성되는 시점에 더불어 생성된다

생성자 함수는

사용자가 직접 정의한 사용자 정의 생성자 함수와

자바스크립트가 기본 제공하는 빌트인 생성자 함수

로 구분할 수 있다.

1. **사용자 정의 생성자 함수와 프로토타입 생성 시점**

   생성자 함수로서 호출할 수 있는 함수, 즉 constructor는 함수 정의가 평가되어 함수 객체를 생성하는 시점에 프로토타입도 더불어 생성된다.

2. **빌트인 생성자 함수와 프로토타입 생성 시점**

   모든 빌트인 생성자 함수는 전역 객체가 생성되는 시점에 생성된다.

   생성된 프로토타입은 빌트인 생성자 함수의 prototype 프로퍼티에 바인딩된다.

객체 생성 방식과 프로토타입의 결정

객체의 생성 방법 : 객체 리터럴, Object 생성자 함수, 생성자 함수, Object.create 메서드, 클래스(ES6)

✔️ 공통점 : 추상 연산 OrdinaryObjectCreate 에 의해 생성된다.

1. **객체 리터럴에 의해 생성된 객체의 프로토타입**

   객체 리터럴에 의해 생성되는 객체의 프로토타입은 Object.prototype이다. 객체 리터럴에 의해 생성된 객체는 Object.prototype을 상속받는다.

```
const obj = { x: 1 };

console.log(obj.constructor === Object); // true
console.log(obj.hasOwnProperty('x')); // true
```

2. **Object 생성자 함수에 의해 생성된 객체의 프로토타입**

   Object 생성자 함수에 의해 생성되는 객체의 프로토타입은 Object.prototype이다. Object 생성자 함수에 의해 생성된 객체는 Object.prototype을 상속받는다.

```
const obj = new Object();
obj.x = 1;

console.log(obj.constructor === Object); // true
console.log(obj.hasOwnProperty('x')); // true
```

3. **생성자 함수에 의해 생성된 객체의 프로토타입**

   생성자 함수에 의해 생성되는 객체의 프로토타입은 생성자 함수의 prototype 프로퍼티에 바인딩되어 있는 객체다.

프로토타입 체인

자바스크립트는 객체의 프로퍼티(메서드 포함)에 접근하려고 할 때 해당 객체에 접근하려는 프로퍼티가 없다면 [[Prototype]] 내부 슬롯의 참조를 따라 자신의 부모 역할을 하는 프로토타입의 프로퍼티를 순차적으로 검색한다.

이를 프로토타입 체인이라 한다.

프로토타입 체인은 자바스크립트가 객체지향 프로그래밍의 상속을 구현하는 메커니즘이다.

Object.prototype

1. 프로토타입의 프로토타입은 언제나 Object.prototype이다.
2. 프로토타입 체인의 최상위에 위치하는 객체는 언제나 Object.prototype이다.
3. 모든 객체는 Object.prototype을 상속받는다.
4. Object.prototype을 프로토타입 체인의 종점이라 한다.
5. Object.prototype의 프로토타입, 즉 [[Prototype]] 내부 슬롯의 값은 null 이다.

프로토타입 체인 vs 스코프 체인

프로토타입 체인 : 상속과 프로퍼티 검색을 위한 메커니즘

스코프 체인 : 식별자 검색을 위한 메커니즘

스코프 체인과 프로토타입 체인은 서로 연관없이 별도로 동작하는 것이 아니라

서로 협력하여 식별자와 프로퍼티를 검색하는 데 사용된다.

오버라이딩과 프로퍼티 섀도잉

- 오버라이딩

  상위 클래스가 가지고 있는 메서드를 하위 클래스가 재정의하여 사용하는 방식
- 프로퍼티 섀됴잉

  상속 관계에 의해 프로퍼티가 가려지는 현상

❗ 하위 객체를 통해 프로토타입의 프로퍼티를 변경 또는 삭제하는 것은 불가능하다. 다시 말해 하위 객체를 통해 프로토타입에 get 액세스는 허용되나 set 액세스는 허용되지 않는다.

프로토타입의 교체

프로토타입은 임의의 다른 객체로 변경할 수 있다.

이것은 부모 객체인 프로토타입을 동적으로 변경할 수 있다는 것을 의미한다.

프로토타입은

1.생성자 함수

2.인스턴스

에 의해 교체할 수 있다.

1. **생성자 함수에 의한 프로토타입의 교체**

   프로토타입으로 교체한 객체 리터럴에는 constructor 프로퍼티가 없다.

   프로토타입을 교체하면 constructor 프로퍼티와 생성자 함수 간의 연결이 파괴된다.

```
Person.prototype = {
  sayHello() {
    console.log(`Hi! My name is ${this.name}`);
  }
};

const me = new Person('Lee');

console.log(me.constructor === Object); // true
```

2. **인스턴스에 의한 프로토타입의 교체**

   인스턴스의 **proto** 접근자 프로퍼티를 통해 프로토타입을 교체할 수 있다.

   proto 접근자 프로퍼티를 통해 프로토타입을 교체하는 것은 이미 생성된 객체의 프로토타입을 교체하는 것이다.

```
function Person(name) {
  this.name = name;
}
const me = new Person('Cho');
const parent = {
  sayHello() {
    console.log(`Hi! My name is ${this.name}`);
  }
};
// me 객체의 프로토타입을 parent로 교체한다.
Object.setPrototypeOf(me, parent);
// 위 코드는 아래의 코드와 동일하게 동작한다.
me.__proto__ = parent;
```

차이점은?

생성자 함수 : Person 생성자 함수의 prototype 프로퍼티가 교체된 프로토타입을 가리킨다.

인스턴스 : Person 생성자 함수의 prototype 프로퍼티가 교체된 프로토타입을 가리키지 않는다.

결론

프로토타입 교체를 통해 상속관계를 동적으로 변경하는것은 까다롭다 → 하지말자

instanceof 연산자

```
객체 instanceof 생성자 함수
```

우변의 생성자 함수의 prototype에 바인딩된 객체가 좌변의 객체의 프로토타입 체인 상에 존재하면 true로 평가되고, 그렇지 않은 경우에는 false로평가된다.

단

instanceof 연산자는

프로토타입의 constructor 프로퍼티가 가리키는 생성자 함수를 찾는 것이 아니라

생성자 함수의 prototype에 바인딩된 객체가 프로토타입 체인 상에 존재하는지 확인한다.

직접 상속

1. **Object.create에 의한 직접 상속**

   Object.create 메서드도 다른 객체 생성 방식과 마찬가지로 추상 연산 OrdinaryobjectCreate를 호출한다.

   첫 번째 매개변수 : 생성할 객체의 프로토타입으로 지정할 객체를 전달

   두 번째 매개변수 : 생성할 객체의 프로퍼티 키와 프로퍼티 디스크립터 객체로 이뤄진 객체 전달

```
/**
* 지정된 프로토타입 및 프로퍼티를 갖는 새로운 객체를 생성하여 반환한다.
* @param {Object} prototype - 생성할 객체의 프로토타입으로 지정할 객체
* @param {Object} [propertiesObject] - 생성할 객체의 프로퍼티를 갖는 객체
* @returns {Object} 지정된 프로토타입 및 프로퍼티를 갖는 새로운 객체
*/
Object.create(prototype[, propertiesObject])
```

장점

1. new 연산자가 없이도 객체를 생성할 수 있다.
2. 프로토타입을 지정하면서 객체를 생성할 수 있다.
3. 객체 리터럴에 의해 생성된 객체도 상속받을 수 있다.

object.prototype 을 직접 호출 하면 안되는 이유

Object.create를 통해 프로토타입 체인의 종점에 위치하는 객체를 생성할 수 있음

체인의 종점에 위치하는 객체는 빌트인 메서드를 사용할 수 없기 때문이다.

```
const obj = Object.create(null);
obj.a = 1;

console.log(obj.hasOwnProperty('a'));
// TypeError: obj.hasWonProperty is not a function
```

따라서 다음과 같이 간접적으로 호출하는 것이 좋다.

```
const obj = Object.create(null);
obj.a = 1;

console.log(Object.prototype.hasOwnProperty.call(obj, 'a')); // true
```

2. **객체 리터럴 내부에서 __proto__에 의한 직접 상속**

   Object.create 메서드는 여러 장점이 있지만 두 번째 인자로 프로퍼티를 정의하는 것은 번거롭다.

   ES6에서는 객체 리터럴 내부에서 __proto__ 접근자 프로퍼티를 사용하여 직접 상속을 구현할 수 있다.

```
const myProto = {x: 10};
const obj = {
    y: 20,
    // 객체를 직접 상속받는다.
    // obj -> myProto -> Object.prototype -> null
    __proto__: myProto
};

console.log(obj.x, obj.y);  // 10 20
```

정적 프로퍼티 / 메서드

정적 프로퍼티/메서드는 생성자 함수로 인스턴스를 생성하지 않아도 참조/호출할 수 있는 프로퍼티/메서드를 말한다.

```
function Person(name) {
  this.name = name;
}
Person.staticProp = 'static prop'; // 정적 프로퍼티
Person.staticMethod = function () { // 정적 메서드
```

```
  console.log('staticMethod');
};
```

정적 프로퍼티/메서드는 인스턴스의 프로토타입 체인에 속한 객체의 프로퍼티/메서드가 아니므로 인스턴스로 접근할 수 없다.

this를 사용하지 않을 경우 정적메서드로 변경 할수 있는데 여기서 this는 생성할 인스턴스를 가리킴

프로퍼티 존재 확인

in 연산자

in 연산자는 객체 내에 특정 프로퍼티가 존재하는지 여부를 확인한다.

```
const person = {
  name: 'Lee',
  address: 'Seoul',
};
console.log('name' in person); // true
console.log('age' in person); // false
console.log('toString' in person); // true
✨ ES6 문법
console.log(Reflect.has(person, 'name')); // true
```

in 연산자는 확인 대상 객체의 프로퍼티뿐만 아니라 확인 대상 객체가 상속받은 모든 프로토타입의 프로퍼티를 확인하므로 주의가 필요하다.

Object.prototype.hasOwnProperty 메서드

객체에 특정 프로퍼티가 존재하는지 확인할 수 있다.

인수로 전달받은 프로퍼티 키가 객체 고유의 프로퍼티 키인 경우에만 true를 반환하고 상속받은 프로토타입의 프로퍼티 키인 경우 false를 반환한다.

```
const person = { name: 'Lee' };
console.log(person.hasOwnProperty('name')); // true
console.log(person.hasOwnProperty('toString..')); // false
```

프로퍼티 열거

for...in 문

객체의 모든 프로퍼티를 순회하며 열거하려면 for...in문을 사용한다.

```
const person = {
  name: 'Lee',
  address: 'Seoul',
};
for(const key in person) {
  console.log(key + ': ' + person[key]);
}
// name: Lee
// address: Seoul
```

- for...in 문은 in 연산자처럼 순회 대상 객체의 프로퍼티뿐만 아니라 상속받은 프로토타입의 프로퍼티까지 열거한다. 하지만 상속받은 프로토타입의 프로퍼티 어트리뷰트 [[Enumeralbe]]의 값이 false이면 열거하지 않는다.
- for...in문은 프로퍼티 키가 심벌인 프로퍼티는 열거하지 않는다

Object.keys/values/entries 메서드

- Object.keys 메서드객체 자신의 열거 가능한 프로퍼티 키를 배열로 반환한다.

```
const person = {
  name: 'Lee',
  address: 'Seoul',
  \__proto\__: { age: 20 }
```

```
};
console.log(Object.keys(person)); // ["name", "address"]
```

- Object.values 메서드 (ES8)객체 자신의 열거 가능한 프로퍼티 값을 배열로 반환한다.

```
console.log(Object.values(person)); // ["Lee", "Seoul"]
```

- Object.entries 메서드 (ES8)객체 자신의 열거 가능한 프로퍼티 키와 값의 쌍의 배열을 배열에 담아 반환한다.

```
console.log(Object.entries(person));
// [["name", "Lee"], ["address", "Seoul"]]
```

Reference

- https://velog.io/@ywc8851/모던자바스크립트-19장-프로토타입
- https://velog.io/@kozel/모던-자바스크립트-19장-프로토타입#19112-객체-리터럴-내부에서-__proto__에-의한-직접-상속